Nombre y Apellido	LU	NO

Corrector:	TOMÁS		
Nota Final / Ejs:	1	2	3
A	P	P	P

Algoritmos y Estructuras de Datos II
Segundo recuperatorio — 8 de Julio de 2022

Aclaraciones

- El parcial es a libro casi-cerrado. Solo es posible tener impreso el teorema maestro y el apunte de módulos básicos. Además, pueden tener una hoja (2 carillas), escrita a mano, con los apuntes que se deseen.
- Cada ejercicio debe entregarse **en hojas separadas**. Las mismas deben estar numeradas.
- Incluir en esta hoja: nombre, apellido, el número de orden asignado, número de libreta.
- Incluir en cada hoja entregada: nombre, apellido y número de libreta.
- Cada ejercicio se calificará con **P**erfecto, **A**probado, **R**egular, o **I**nsuficiente.
- El parcial estará aprobado si las notas de los tres ejercicios tienen al menos dos **A**.
- Los ejercicios **no** se recuperan por separado.
- Se encuentran disponibles para utilizar todas las estructuras de datos presentadas en la teórica con las operaciones y complejidades dadas en las mismas.

Ej. 1. Ordenamiento

Diremos que un arreglo es semi ordenado para L y H (dos números naturales, con $L < H$) si cumple que todos los valores menores o iguales a L aparecen en orden decreciente; todos los valores mayores o iguales a H aparecen en orden creciente; y la cantidad de elementos que hay en el rango (L, H) es a lo sumo $\sqrt(n)$ siendo n el tamaño del arreglo.

Ejemplo: La secuencia $S = [3, 17, 2, 20, 1, 23, 5, 11, 9]$ está semi ordenada para $L = 4$ y $H = 12$, pues los valores menores a 4 aparecen en orden 3, 2, 1 y los valores mayores a 12 aparecen en orden 17, 20, 23.

Formalmente:

- $(\forall i, j : \mathbb{Z})\ 0 \leq i \leq j < n \Rightarrow (A[i] \leq L \wedge A[j] \leq L \Rightarrow A[i] \geq A[j]) \wedge (A[i] \geq H \wedge A[j] \geq H \Rightarrow A[i] \leq A[j])$
- $\#\{x : \mathbb{Z} | x \in A \wedge L < x < H\} < \sqrt{n}$

a) Proponga un algoritmo de ordenamiento $ordenarSemi(A : arreglo(nat), L : nat, H : nat)$ de complejidad $O(n)$. Puede en este punto utilizar (sin tener que definirlos) cualquier de los algoritmos de ordenamiento definidos en la materia, indicando claramente la complejidad de su aplicación.

b) Justifique detalladamente la correctitud del algoritmo y su complejidad temporal.

Ej. 2. Elección de estructuras

Se tiene un arreglo R con n strings sin repeticiones que define un *ranking*. Se tiene además un arreglo A de m strings tal que todos ellos aparecen en el ranking R. Se quiere ordenar el arreglo A en función del ranking definido por R. Es decir, dados dos elementos s y t de A, s será "menor" que t, si aparece en R antes que t. Por ejemplo, si tenemos

$R =$ `[Brasil, Argentina, Alemania, Chile, Colombia, Francia]` y
$A =$ `[Chile, Francia, Brasil, Chile, Argentina, Brasil]`.

entonces el orden correcto para A sería `[Brasil, Brasil, Argentina, Chile, Chile, Francia]`.

Suponiendo que el largo de todos los strings está acotado por una constante y que el abecedario también lo está, proponga un algoritmo de ordenamiento que resuelva el problema en una complejidad $O(n + m)$, donde n y m son las cantidades de elementos en el ranking y en el arreglo a ordenar, respectivamente. Justifique la correctitud del algoritmo y su complejidad temporal. **TIP**: considerar almacenar el Ranking en una estructura auxuliar como ser AVL, TRIE o HEAP.

Ej. 3. Divide and Conquer

Se tienen un arreglo A de n números enteros (con n mayor a 4 y potencia de 2) que se encuentra ordenado de manera estrictamente decreciente hasta un cierto punto y luego de manera estrictamente creciente. Se pide encontrar el mínimo del arreglo.

Por ejemplo, para $A = [4, 2, -3, 5]$ el resultado es -3 y para $A = [4, 3, 6, 9, 10, 11, 13, 14]$ el resultado será 3.

Formalmente:

- $(\exists x : \mathbb{Z})\ x \geq 2 \land n = 2^x$
- $(\exists i : \mathbb{Z})\ (0 \leq i < n)\ \land ((\forall j : \mathbb{Z})\ 0 \leq j < n \land\ (j < i \Rightarrow A[j] > A[j+1]) \land\ (j > i \Rightarrow A[j-1] < A[j]))$

a) Implementar la función: min(**in** A: arreglo(nat)) $\rightarrow$ nat que resuelve el problema planteado. La función debe ser de tiempo $O(\log n)$.

b) Calcular y justificar la complejidad del algoritmo propuesto utilizando el teorema maestro.

HOJA N° 1/3

FECHA

1) arreglo(Nat) ordenarSemi(arreglo(Nat) A, Nat L, Nat H){ pre:{esSemi(A,L,H)}

```
arreglo(Lista(Nat)) aux(3);
for(int i=0; i<|A|; i++){
    if(A[i] ≤ L){
        aux[0] = AgregarAdelante(aux[0], A[i])          // O(1)
    }else{
        if(A[i] ≥ H){
            aux[2] = AgregarAtras(aux[2], A[i]);        // O(1)
        }else{
            aux[1] = AgregarAtras(aux[1], A[i]);        // O(1)
        }
    }
}                                                       O(n)
arreglo(Nat) menorIgualL = list2Array(aux[0]);          // O(|aux[0]|)
arreglo(Nat) entreLH = list2Array(aux[1]);              // O(|aux[1]|)
arreglo(Nat) mayorIgualH = list2Array(aux[2]);          // O(|aux[2]|)
Mergesort(entreLH);                                     // O(√n log √n)
arreglo(Nat) res = menorIgualL + entreLH + mayorIgualH; // O(n)
return res;
}
```

+ = concatenar

b) Correctitud y complejidad

El algoritmo se basa en la idea de que en el arreglo se pueden encontrar 3 grupos, de los cuales sabemos que dos estan ordenados. Aprovechando esto, los dos ordenados se insertan en unas listas de forma que ambas queden ordenadas de manera creciente (en particular, la de aquellos ≤L).

La 3° lista (aux[1]) no presenta orden, por lo cual es necesario ordenarla. Es importante aclarar que, dado que su tamaño es como mucho $\sqrt{n}$, todos los algoritmos de ordenamiento hasta $O(n^2)$ sirven (ya que $O(\sqrt{n}^2) = O(n)$).

Finalmente, se concatenan los 3 arreglos ya ordenados.

NOTA

Por lo dicho anteriormente:

- Armar las listas: $O(n)$
- crear los arreglos: O(tamaño de la lista), eventualmente, $O(n)$
- ordenar el arreglo de elementos entre L y H:

  hasta $O(n)$ *

- Unir los arreglos será $O(n)$

* si

$O(n \log n) < O(n^2)$

$\Rightarrow O(\sqrt{n} \log \sqrt{n}) < O((\sqrt{n})^2) = O(n)$

HOJA N° 2/3

FECHA

3)
```
Nat min(arreglo(Nat) A){        {|A| = 2^j , j>1}
  Nat res = minAux(A, 0, |A|-1);
  return res;
}

Nat minAux(arreglo(Nat)& A, Nat inicio, Nat fin){
  Nat mitad = inicio + (fin-inicio)/2;
  if (fin - inicio < 2){
      if (A[fin] > A[inicio]){
        return A[inicio];
      }else{
        return A[fin];
      }
  }else{
      if (A[mitad] < A[mitad+1]){
        return minAux(A, inicio, mitad);
      }else{
        return minAux(A, mitad+1, fin);
      }
  }
}
```

[10 9 8 7 6 5 2 3]

3, 1

b)
$T(n) = a\,T(\frac{n}{c}) + f(n)$, a = cantidad de subproblemas , $\frac{n}{c}$ = tamaño subproblemas

f(n) = demas operaciones

FALTA CASO BASE

$\Rightarrow T(n) = T(\frac{n}{2}) + \Theta(1)$

entonces quiero ver que, por Teorema Maestro, $T(n) \in O(\log n)$

$\Rightarrow$ 2do caso del Teorema:

$f(n) = \Theta(n^{\log_c a}) \equiv \Theta(n^{\log_2 1}) \equiv \Theta(n^0) \equiv \Theta(1)$

Por lo tanto:

$T(n) = \Theta(n^{\log_c a} \log n) \equiv \Theta(n^{\log_2 1} \log n)$

$\equiv \Theta(n^0 \log n)$

$\equiv \Theta(\log n)$

NOTA

2)

```
arreglo(string) ranking(arreglo(string) R, arreglo(string) A){

    DiccTrie(Nat) apariciones;   (*)

    for(int i=0; i<|R|; i++){                                   } O(n)
        definir(apariciones, R[i], 0);                  O(1)
    }
    for(int i=0; i<|A|; i++){                                   } O(m)
        definir(apariciones, A[i], significado(apariciones, A[i]) +1);   O(1)
    }
    arreglo(string)       |A|);
    int posRes=0;
    for(int i=0; i<|R|; i++){                                   } O(n+m)

        int significado = significado(apariciones, R[i]);
        for(int j=0; j<significado; j++){

            res[posRes] = R[i];                         O(1)
            posRes++;
        }
    }
    return res;
}
```

(*) un Trie está definido como un arbol en donde cada nodo tiene hasta 256 siguientes nodos y su posición representa una letra en la cual puede haber, o no, una definición. Por eso solo se define la definicion, ya que implicitamente representa una palabra que es la unión de la letra que representa en cada nodo para llegar a él.

Correctitud y Complejidad;

Los strings al estar acotados, así como la cantidad de letras en el abecedario, y como vimos en clase; permiten a las operaciones de inserción y búsqueda ser O(1).

La creación del Trie cuesta O(n) (con las claves pertenecientes a R) y definir los significados cuesta O(m) (con claves de A).

NOTA

Finalmente, la reconstrucción del arreglo cuesta $O(n+m)$ porque se accede y se obtiene los significados del Trie n-veces y se insertan en el arreglo a retornar un total de m-veces a lo largo de todas esas n-veces. ✓

De esta forma las complejidades quedan

Crear Trie = $O(n)$

Re-Definir significados = $O(m)$

Reconstruir arreglo = $O(n+m)$ ✓

EXCELENTE PARCIAL!